Sabanci University, CS406

HW01 - Parallelization of Sinkhorn-Knopp Algorithm

Kaya Gokalp

March 28, 2021

1 Introduction

In this homework we were asked to parallelize Sinkhorn-Knopp algorithm which is used do construct D_C and D_R matrices such that for input matrix A,

$$D_C A D_R$$

is doubly scholastic. Doubly scholastic means sum of any column or row is equal to 1. The input, matrix A needs to be in some form to be able to find the answer but in this homework input matrix A is assumed to be satisfying those requirements. After implementing the algorithm correctly, parallelization techniques applied and results are discussed in following sections.

2 Preprocessing

Sparse matrix representation using CCS and CRS: Since we are dealing with large matrices moving and operating on the whole 2-dimensional matrix array is too expensive, we are using a representation for our sparse matrices CCS and CRS. In this homework construction of the required arrays for these representations are done for us. So the pre-processing step is already done. Only initialization of d_r and d_c to 1 is added to preprocessing step.

3 Algorithm

3.1 Using CCS and CRS representations in the algorithm

In the algorithm we are using a_{ij} for calculating rsum (sum of row) and csum (sum of columm). Since we need to do multiplication between a_{ij} and d_R[i] or d_C[j] we need to somehow access a_{ij}. A simple trick is used for this purpose. Since all the elements in the matrices are either 0 or 1, the multiplication for all i or j simple becomes a summation of the d_R[i] or d_c[j] where the corresponding a_{ij} is non-zero. So basically to be able to calculate rsum or csum the position of the data in the currently iterating row or column is needed. This is obtained by looping from xadj[i] to xadj[i+1] while reaching the value of adj with corresponding loop index (for column same procedure but txadj and tadj). To demonstrate it more clearly this is the corresponding code block for row sum (where i goes from row 0 to row n):

```
int rowIndex = xadj[i];
for(;rowIndex<xadj[i+1]; rowIndex++)
{
  rsum += cv[adj[rowIndex]];
}
```

Corresponding code block for column sum (where i goes from column 0 to column n):

```
int colIndex = txadj[i];
for(;colIndex<txadj[i+1]; colIndex++)
{
  csum += rv[tadj[colIndex]];
}
```

Assuming the example matrix from Figure 3.1 is the input. An example run for the code block above goes like the following.

Row sum for row 0:

1. i = 0, rowIndex = xadj[0] which is 0
2. xadj[1] which is 3 so the loop will run from 0 to 3.
3. adj[0] = 0 which is the first non zero value's column in row 0, similarly adj[1] is 2 and adj[2] is 3. Similarly adj[1] and adj[2] is the second and third non zero value from row 0
4. Since a_{ij} is 1 when it is non-zero and from step3 non-zero valued indices are found in the row 0. So summation of d_j[adj[0]], of d_j[adj[1]] and of d_j[adj[2]] is our rsum. Which is 1+1+1 in the first run assuming d_j is all 1's.

Column sum for row 0:

1. i = 0, colIndex = txadj[0] which is 0
2. txadj[1] which is 2 so the loop will run from 0 to 2.
3. tadj[0] = 0 which is the first non zero value's row in col 0, similarly tadj[1] is 4.
4. Since a_{ij} is 1 when it is non-zero and from step3 non-zero valued indices are found in the col 0. So summation of d_i[tadj[0]] and d_i[tadj[1]] is the csum. Which is 1+1 in the first run assuming d_i is all 1's.

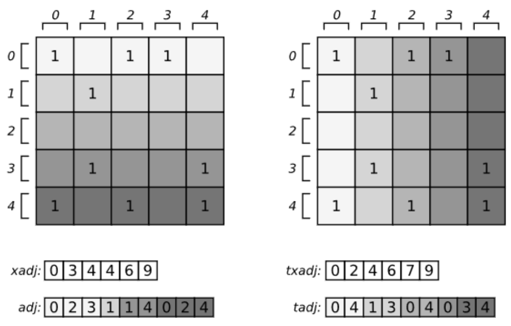

Figure 3.1: Example matrix with xadj, adj, txadj, tadj values

3.2 Calculating the error per iteration

WHILE CALCULATING THE ERROR for each row i, for any non-zero valued index a, d_C[a] is multiplied with d_R[i] and summed up. The resulting value should be 1 so 1 - the sum result is our error. The biggest error is selected as the error for the iteration. The following code block is used to calculate error for the iteration.

```
#pragma omp parallel for schedule(guided) reduction(max \
                                                   : maxErrorSum)
    for (int i = 0; i < nov - 1; i++)
    {
      double errorSum = 0;
      double errorValue = 0;
      int rowIndex = xadj[i];
      for (; rowIndex < xadj[i + 1]; rowIndex++)
      {
        errorSum += cv[adj[rowIndex]] * rv[i];
        errorValue = abs(1.0 - errorSum);
      }
      if (errorValue > maxErrorSum)
      {
        maxErrorSum = errorValue;
      }
    }
    std::cout << "iter " << it << " - error " << (maxErrorSum) << "\n";
  }
```

4 RUN TIMES AND PERFORMANCE

4.1 Before any optimization - Serial Error Check

THE TABLE BELOW shows the un-optimized version of the code written for this homework. This code uses basically #pragma omp parallel for before loops of the algorithm. No further optimization exists on this version. The error check is not parallelized. (Table 4.1)

Optimization Level	**1 thread**	**2 thread**	**4 thread**	**8 thread**	**16 thread**
O_0	6.92	5.40	4.49	3.97	3.72
O_1	1.58	1.25	0.94	0.81	0.78
O_2	1.52	1.15	0.89	0.76	0.73
O_3	1.58	1.16	0.88	0.74	0.73

Table 4.1: No optimization. Runtimes in seconds (cage15)

4.2 Before any optimization (to algorithm)- Parallel Error Check

THE TABLE BELOW shows the performance of this code which is constructed in the following way. It uses basically #pragma omp parallel for before loops of the algorithm. No further optimization exists on this version. The error check is parallelized (different from the section 4.2). (Table 4.1)

Optimization Level	**1 thread**	**2 thread**	**4 thread**	**8 thread**	**16 thread**
O_0	6.91	3.91	2.13	1.12	0.6
O_1	1.62	0.97	0.59	0.34	0.27
O_2	1.60	0.99	0.58	0.34	0.29
O_3	1.59	0.98	0.57	0.34	0.27

Table 4.2: No optimization to algorithm, error check is parallel. Runtimes in seconds (cage15)

4.3 After some optimization (to algorithm, schedule-dynamic)- Parallel Error Check

THE TABLE BELOW shows the results after some optimization. This code uses #pragma omp parallel for before loops of the algorithm also dynamic schedule is set for the same for loops The chunk size is 8. My initial expectations for this No further optimization exists on this version. The error check is parallelized, but not optimized. (Table 4.3)

Optimization Level	**1 thread**	**2 thread**	**4 thread**	**8 thread**	**16 thread**
O_0	7.04	4.87	2.99	1.9	1.48
O_1	1.75	1.71	1.65	1.39	1.28
O_2	1.76	1.77	1.58	1.38	1.23
O_3	1.66	1.70	1.45	1.34	1.27

Table 4.3: Optimization : schedule(dynamic, chunk-size=8), error check is parallel. Runtimes in seconds (cage15)

4.4 After some optimization (to algorithm, schedule-guided)- Parallel Error Check

THE TABLE BELOW shows the results after some optimization. Like in the previous section. This code uses #pragma omp parallel for before loops of the algorithm also guided schedule is set for the same for loops. No further optimization exists on this version. The error check is parallelized, but not optimized. (Table 4.4)

Optimization Level	**1 thread**	**2 thread**	**4 thread**	**8 thread**	**16 thread**
O_0	6.93	3.72	1.95	1.01	0.53
O_1	1.65	0.92	0.51	0.31	0.27
O_2	1.66	0.95	0.51	0.33	0.27
O_3	1.53	0.87	0.48	0.32	0.27

Table 4.4: Optimization : schedule(guided), error check is parallel. Runtimes in seconds (cage15)

4.5 With optimization

THE TABLE BELOW shows the optimized version of the code written for this homework. This code uses #pragma omp schedule(guided) (for algorithm and error check) and also reduction (for the error check). Error summation part is also parallelized. (Table 4.5)

Optimization Level	**1 thread**	**2 thread**	**4 thread**	**8 thread**	**16 thread**
O_0	6.89	3.47	1.76	0.88	0.57
O_1	1.53	0.82	0.45	0.29	0.26
O_2	1.64	0.91	0.46	0.29	0.25
O_3	1.64	0.87	0.46	0.30	0.25

Table 4.5: Optimization : schedule(guided),reduction used, error check is parallel. Runtimes in seconds (cage15)

4.6 Optimized vs Non-Optimized with different inputs

TABLES BELOW shows the optimized version of the code compared to the un-optimized version with different inputs. Table 4.6 is the un-optimized version of the code. Same inputs and compiler flag (O_3) also used with optimized version which can be seen Table 4.7.

Input	**1 thread**	**2 thread**	**4 thread**	**8 thread**	**16 thread**
Hamrle3	0.54	0.38	0.17	0.16	0.15
atmosmodl	0.8	0.42	0.20	0.13	0.17
vas_stokes	2.67	1.89	1.47	0.95	0.52
cage15	7.96	4.96	2.87	1.89	2.15
stokes	27.77	19.38	14.46	7.81	5.38

Table 4.6: O_3 optimization flag used. Un-optimized code

Input	**1 thread**	**2 thread**	**4 thread**	**8 thread**	**16 thread**
Hamrle3	0.53	0.32	0.16	0.11	0.11
atmosmodl	0.81	0.42	0.22	0.15	0.14
vas_stokes	2.75	1.52	0.97	0.55	0.43
cage15	8.22	4.34	2.31	1.59	1.25
stokes	29.26	15.67	8.66	5.65	4.55

Table 4.7: O_3 optimization flag used. Optimized code

5 Conclusion

At the beginning of this homework , I started with an axiom which is the idea of not re constructing the matrices from their CCS and CRS representations. I made this assumption because reconstructing the whole matrix would be an expensive operation. I built up my trials on that assumption. Further than that throughout this homework some of my initial ideas were proved wrong with the test I made. For example the Section 4.3. At section 4.3 I tried scheduling by chunks of 8 since the size of a double is 8 bytes and the cache line of the gandalf is 64 bytes. So I tried to overcome false sharing but it made it worse and the program lose it's scalability with respect to #threads. The optimizations I made increased the time spent in single thread configuration. This was something I was excepting because in order to be able to scale the program better with threads some of the operations creates overhead which causes the specified effect. I found out that after the optimizations the program is scaling better. But the way I was dividing the work is not the best solution to this problem. Since I was basically dividing each row and column to a thread and the amount of non zeros in rows and columns are not equal to each other. So the work was not divided equally. It was divided but since it was not equally divided my program's scaling performance is a little bit more effected than I wanted it to be from the input. If the input is equally divided into rows and columns my program was close to linear scaling until 16 threads. Ideally I would be dividing the rows and columns with respect to their #non zeros. But the time was limited and I couldn't prepare it in time so I used a guided scheduling which basically starts the chunk size big and decreases it until work is properly balanced. This is an solution to the unequal amount of work problem I mentioned but amount of the overhead introduced to the result from guided scheduling is bigger than dynamic scheduling. Also the calculation of the error is corrected after some time with an absolute value, until then my runtimes were around %40 faster than the current status. I used abs() from <math.f>. After the deadline of the homework i will be testing and finding out the reason behind this slowdown whether it is from abs() or anything else.